Requisições condicionais HTTP

O HTTP possui o conceito de **requisições condicionais**, onde o resultado, e até mesmo o sucesso de uma requisição, pode ser controlado comparando os recursos afetados com um validador. Essas requisições são úteis para validar conteúdo em cache, garantindo que ele só seja buscado se for diferente da cópia que já está disponível no navegador.

Requisições condicionais também são úteis para garantir a integridade de um documento ao retomar um download ou para evitar atualizações perdidas ao fazer upload ou modificar um documento no servidor.

Princípios

Requisições condicionais HTTP são requisições que são executadas de maneira diferente, dependendo do valor de cabeçalhos específicos. Esses cabeçalhos definem uma **pré-condição**, e o resultado da requisição será diferente caso a pré-condição seja satisfeita ou não.

Os comportamentos diferentes são definidos pelo **método da requisição** usada e pelo **conjunto de cabeçalhos** utilizados para a pré-condição:

- Para **métodos seguros**, como GET, que normalmente busca um documento, a requisição condicional pode ser usada para **retornar o documento apenas se necessário**, economizando largura de banda.
- Para **métodos inseguros**, como PUT, que normalmente envia um documento, a requisição condicional pode ser usada para fazer upload **somente se** o original no qual se baseia for o mesmo que está armazenado no servidor.

Validadores

Todos os cabeçalhos condicionais tentam verificar se o recurso armazenado no servidor corresponde a uma versão específica. Para isso, as requisições condicionais precisam indicar a **versão** do recurso. Como comparar o recurso inteiro byte a byte é impraticável (e nem sempre desejado), a requisição transmite um valor que descreve a versão. Esses valores são chamados de **validadores**, e existem dois tipos:

- A data da última modificação do documento: Last-Modified
- Uma string opaca, identificando exclusivamente cada versão: ETag (ou *entity tag*)

Comparar versões do mesmo recurso pode ser complicado. Dependendo do contexto, existem dois tipos de validação:

- **Validação forte**: usada quando se espera uma identidade byte a byte (ex: retomar um download)
- **Validação fraca**: usada quando o agente do usuário só precisa saber se os conteúdos são equivalentes (ex: alterações apenas em anúncios ou rodapé não são relevantes)

Ambos os validadores (Last-Modified e ETag) permitem os dois tipos de validação. O HTTP usa **validação forte por padrão**, mas especifica quando a fraca pode ser usada.

Validação forte

A validação forte garante que o recurso é, byte por byte, idêntico ao que está sendo comparado. Isso é **obrigatório** para alguns cabeçalhos condicionais, e o **padrão** para os outros. É muito rigoroso e pode ser difícil de garantir no lado do servidor, mas garante que não haja perda de dados — às vezes em detrimento do desempenho.

É difícil ter um identificador único para validação forte com Last-Modified. Muitas vezes isso é feito usando um ETag gerado com um hash MD5 do recurso (ou derivado).

Validação fraca

A validação fraca difere da forte porque considera duas versões de um documento como idênticas **se o conteúdo for equivalente**.
Por exemplo, uma página que só difere de outra por causa da data no rodapé ou por anúncios diferentes seria considerada igual com validação fraca, mas **diferente** com validação forte.

Criar um sistema de ETags com validação fraca é muito útil para otimizar desempenho de cache, mas pode ser complexo, pois exige saber quais elementos da página são relevantes.

Cabeçalhos condicionais

Vários cabeçalhos HTTP, chamados de **cabeçalhos condicionais**, conduzem a requisições condicionais. São eles:

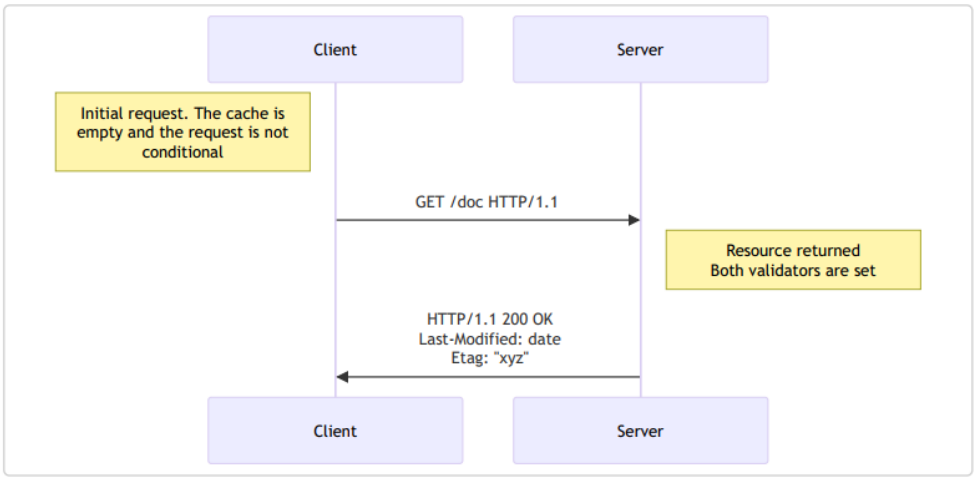
- If-Match: tem sucesso **se o ETag do recurso remoto for igual** a algum dos listados. Realiza **validação forte**.
- If-None-Match: tem sucesso **se o ETag for diferente** de todos os listados. Realiza **validação fraca**.
- If-Modified-Since: tem sucesso se a data Last-Modified do recurso remoto for **mais recente** do que a fornecida.
- If-Unmodified-Since: tem sucesso se a data Last-Modified for **mais antiga ou igual** à fornecida.
- If-Range: semelhante a If-Match ou If-Unmodified-Since, mas só aceita **um único ETag ou data**.
  Se falhar, a requisição de faixa (Range) falha e o servidor retorna um 200 OK com o **recurso completo**, em vez de 206 Partial Content.

Nota: Como uma mudança na codificação do conteúdo exige alteração no ETag, alguns servidores modificam os ETags ao comprimir respostas (ex: proxies reversos).
O Apache, por padrão, adiciona o nome do método de compressão (ex: -gzip) aos ETags, mas isso pode ser configurado com a diretiva DeflateAlterETag.

Casos de uso

Atualização de cache

O caso de uso mais comum para requisições condicionais é a **atualização de cache**.
Com um cache vazio, ou sem cache, o recurso requisitado é enviado de volta com o status 200 OK.

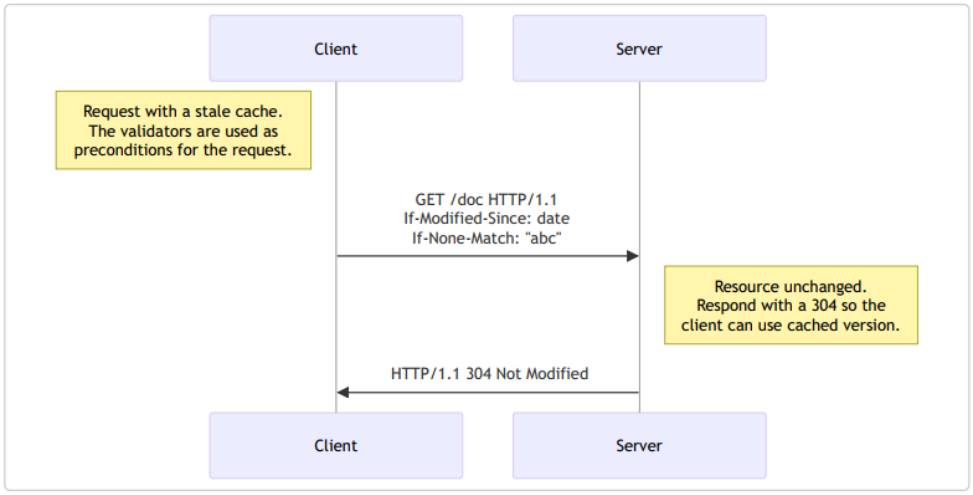


Junto com o recurso, os **validadores** são enviados nos cabeçalhos. Neste exemplo, tanto Last-Modified quanto ETag são enviados, mas poderia ser apenas um deles.

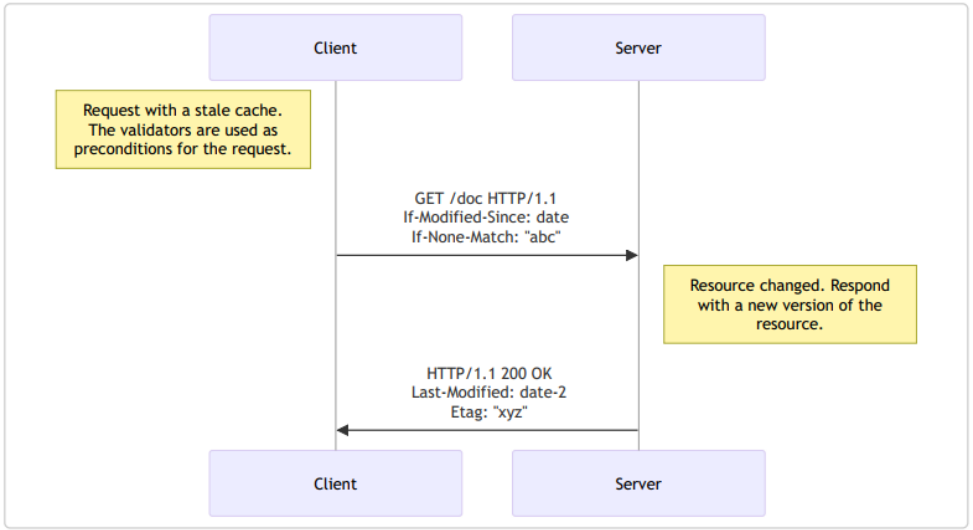
Esses validadores são armazenados em cache com o recurso (como todos os cabeçalhos) e serão usados para montar requisições condicionais, uma vez que o cache fique obsoleto.

Enquanto o cache **não estiver obsoleto**, nenhuma requisição é feita.
Mas uma vez que ele fique obsoleto (isso é geralmente controlado pelo cabeçalho Cache-Control), o cliente **não usa diretamente** o valor em cache, mas em vez disso faz uma **requisição condicional**.
O valor do validador é usado como parâmetro nos cabeçalhos If-Modified-Since e If-None-Match.

Se o recurso **não tiver sido modificado**, o servidor responde com 304 Not Modified.
Isso torna o cache atual novamente, e o cliente usa o recurso em cache.
Embora haja uma troca requisição/resposta que consome recursos, isso é **mais eficiente** do que retransmitir todo o recurso novamente.



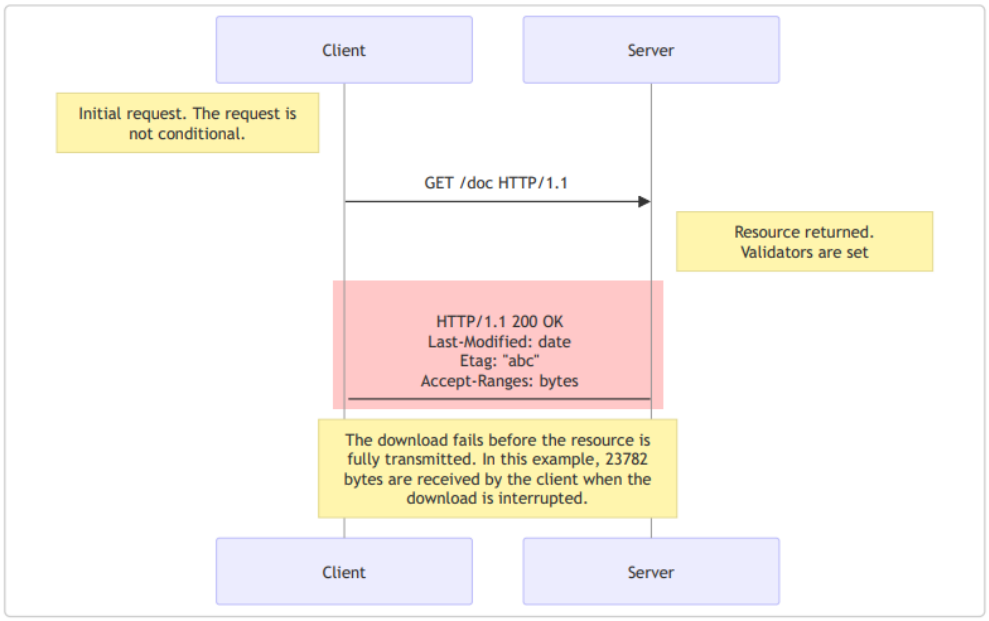
Se o recurso **tiver sido modificado**, o servidor responde com 200 OK e envia a nova versão do recurso (como se a requisição não fosse condicional). O cliente usa o novo recurso e o armazena em cache.



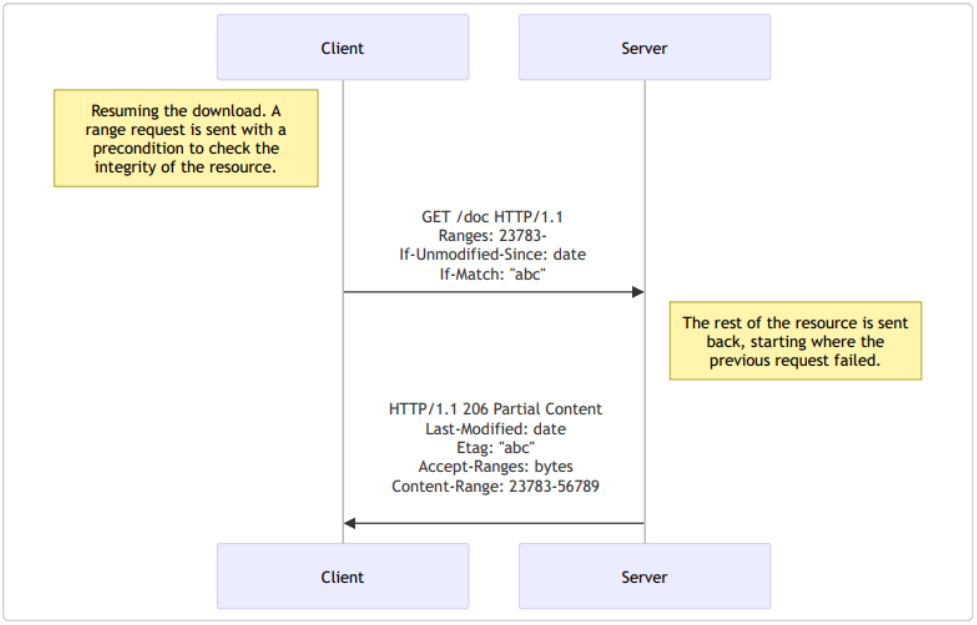
Além da configuração dos validadores no servidor, esse mecanismo é transparente: todos os navegadores gerenciam um cache e enviam esse tipo de requisição condicional **sem que desenvolvedores web precisem fazer nada**.

Integridade de um download parcial

Download parcial de arquivos é uma funcionalidade do HTTP que permite retomar operações anteriores, economizando largura de banda e tempo ao manter os dados já recebidos:



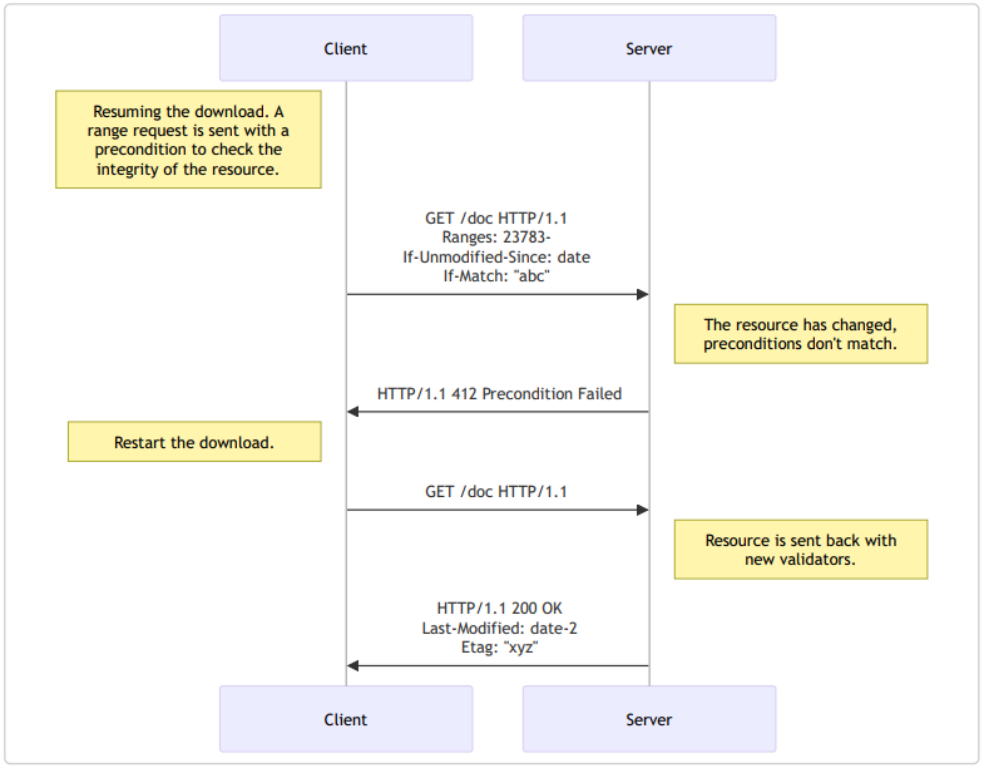
Um servidor que suporta downloads parciais anuncia isso com o cabeçalho Accept-Ranges.
Depois disso, o cliente pode retomar o download enviando o cabeçalho Range com os trechos ausentes:



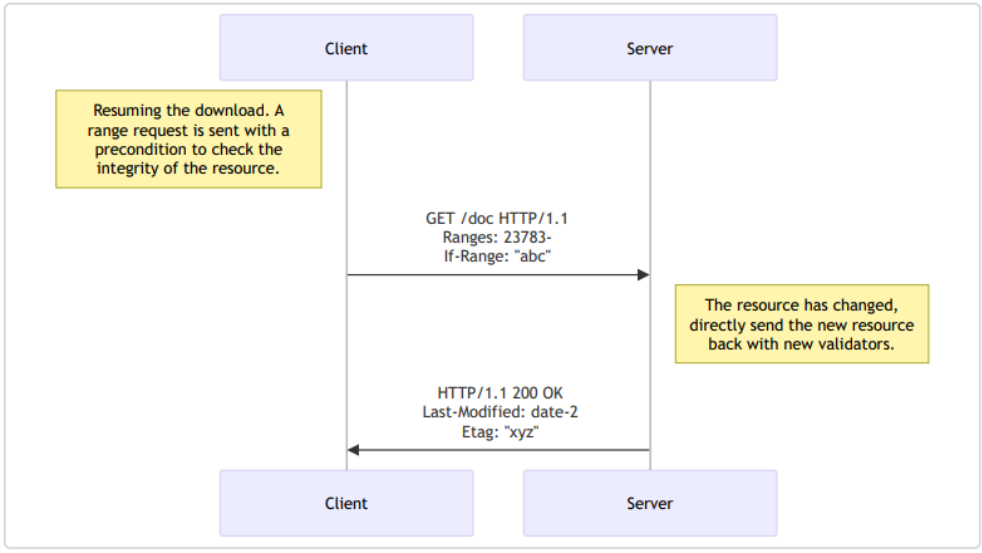
O princípio é simples, mas há um problema potencial:
Se o recurso tiver sido **modificado entre os dois downloads**, os trechos obtidos pertencem a **versões diferentes** do recurso e o documento final será **corrompido**.

Para evitar isso, usam-se requisições condicionais. Para faixas, há duas formas:

- Uma mais flexível: usa If-Unmodified-Since e If-Match.
  Se a pré-condição falhar, o servidor retorna erro e o cliente recomeça o download do início:



Mesmo que funcione, esse método adiciona uma troca extra de requisição/resposta se o documento foi alterado. Isso afeta o desempenho.
O HTTP possui um cabeçalho específico para evitar esse cenário: If-Range.

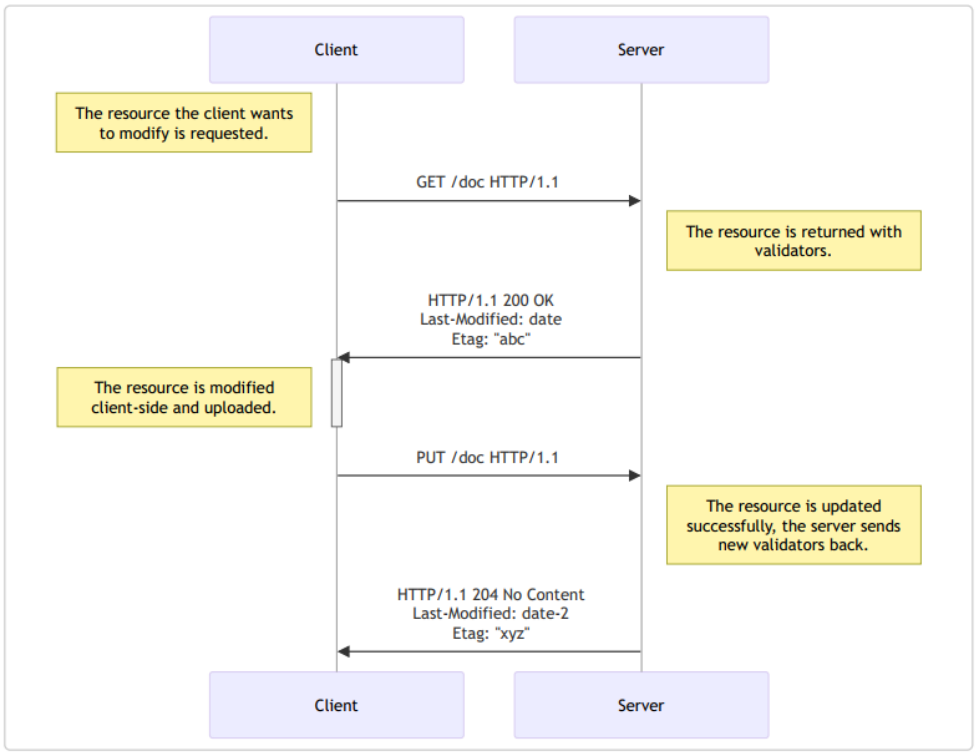


Essa solução é mais eficiente, mas um pouco menos flexível, pois permite **apenas um ETag** como condição. Raramente é necessário mais flexibilidade.

Evitando o problema da atualização perdida com bloqueio otimista

Uma operação comum em aplicações web é **atualizar um documento remoto**.
Isso é comum em qualquer sistema de arquivos ou aplicação de controle de versão, mas qualquer aplicação que permita armazenar recursos remotamente precisa de um mecanismo como esse.
Sites comuns, como wikis e outros CMS, também têm essa necessidade.

Com o método PUT, você pode implementar isso. O cliente primeiro **lê o arquivo original**, modifica-o, e por fim envia de volta para o servidor:

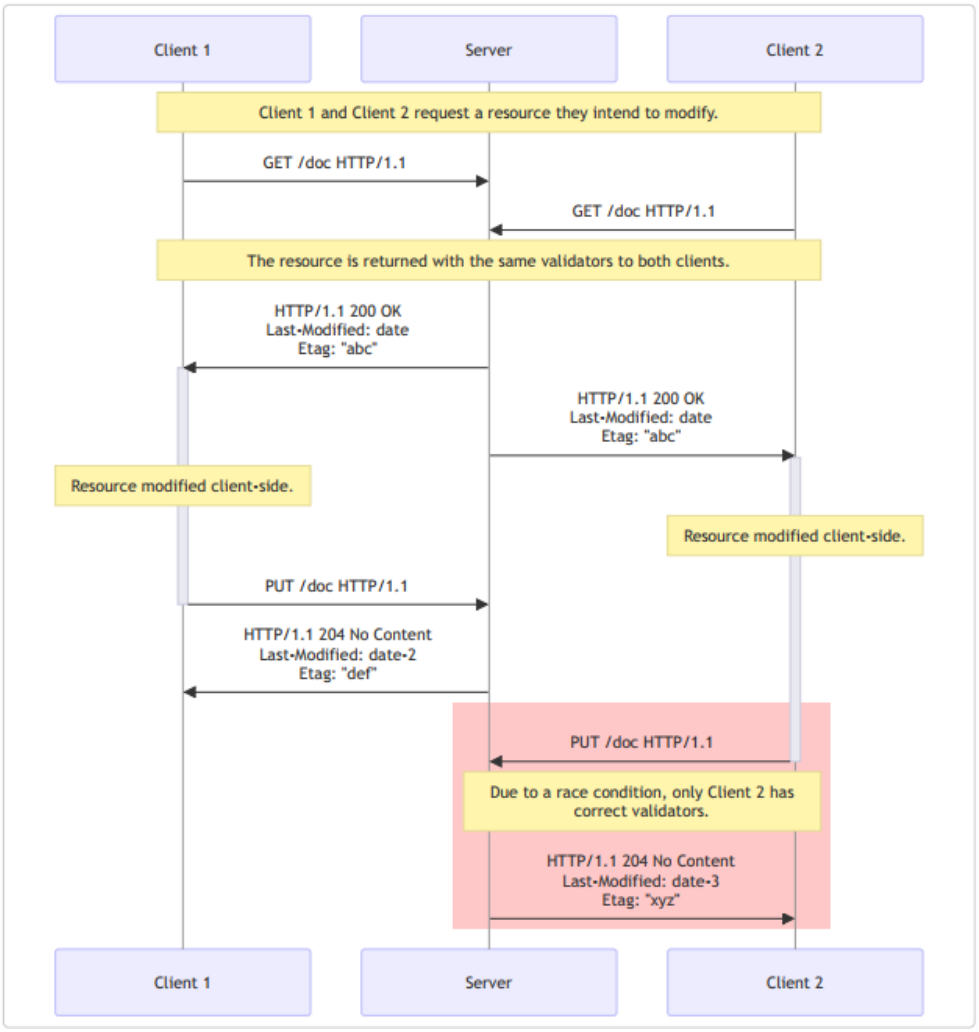


Infelizmente, as coisas ficam imprecisas ao levarmos em conta a **concorrência**.
Enquanto um cliente está modificando sua cópia local do recurso, **outro cliente** pode buscar o mesmo recurso e fazer o mesmo.

O que acontece a seguir é muito ruim:
quando ambos enviam suas alterações ao servidor, as modificações do **primeiro cliente** são **descartadas** pela requisição do segundo cliente, que **não sabe** que o primeiro cliente alterou o recurso.

A decisão de **qual cliente vence** não é comunicada à outra parte.
Qual alteração será mantida varia conforme **a velocidade de envio**, o desempenho dos clientes, do servidor, ou até mesmo da pessoa que edita.
O vencedor muda a cada vez.

Esse é um **problema de condição de corrida (race condition)** e leva a comportamentos problemáticos, difíceis de detectar e depurar:



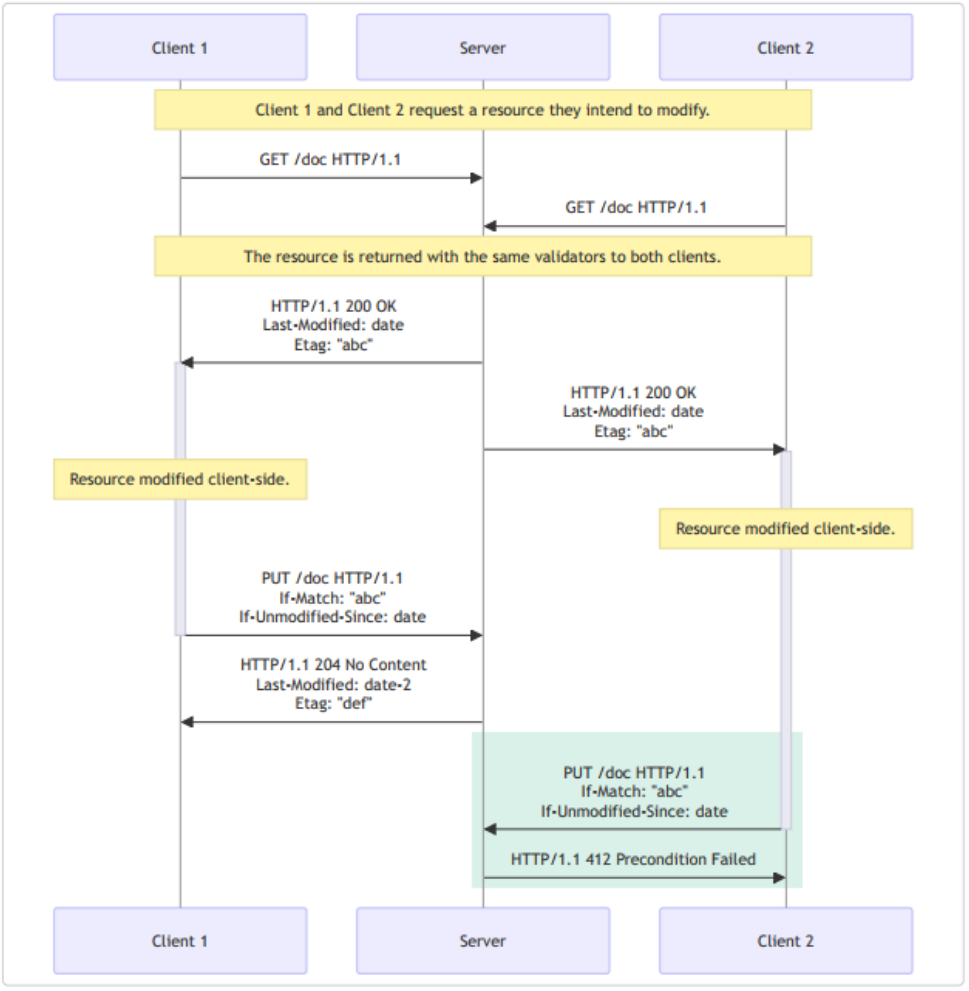
Não há como resolver esse problema sem prejudicar um dos dois clientes.

No entanto, **atualizações perdidas e condições de corrida devem ser evitadas**. Queremos resultados previsíveis e esperamos que os clientes sejam **notificados quando suas alterações forem rejeitadas**.

Implementando bloqueio otimista

Requisições condicionais permitem a implementação do **algoritmo de bloqueio otimista**, usado por muitos wikis ou sistemas de controle de versão.
O conceito é:

1. Permitir que todos os clientes obtenham cópias do recurso.
2. Modificar localmente.
3. Controlar a concorrência permitindo que **apenas o primeiro cliente** envie uma atualização com sucesso.
4. Todas as atualizações subsequentes, baseadas em versões obsoletas, são rejeitadas.



Isso é implementado usando os cabeçalhos If-Match ou If-Unmodified-Since.
Se o ETag **não corresponder** ao arquivo original, ou se o arquivo tiver sido modificado desde que foi obtido, a alteração é rejeitada com um erro **412 Precondition Failed**.

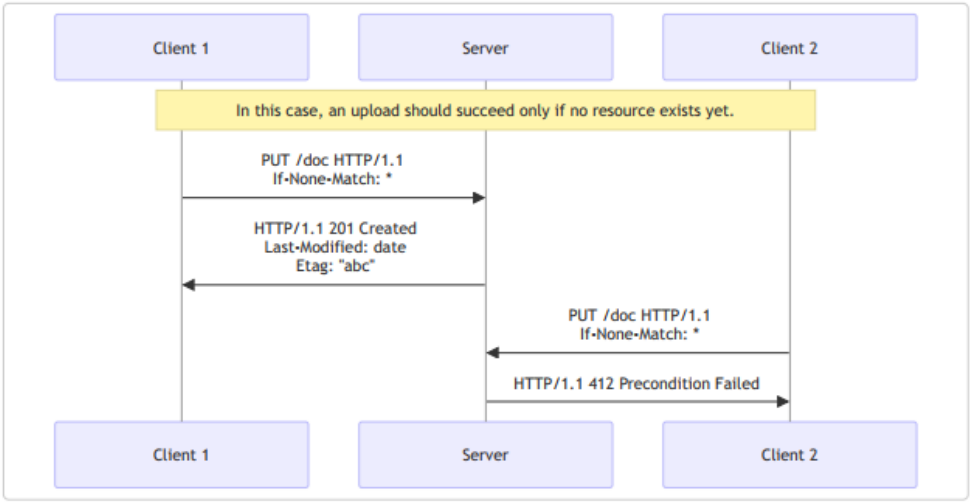
Cabe então ao cliente lidar com o erro:

- notificando o usuário para tentar novamente com a nova versão, ou
- exibindo um **diff** entre as versões, ajudando o usuário a decidir quais mudanças manter.

Lidando com o primeiro envio de um recurso

O primeiro envio (*upload*) de um recurso é um caso especial do cenário anterior.
Como qualquer atualização de um recurso, ele está sujeito a uma **condição de corrida** se dois clientes tentarem fazer o envio **ao mesmo tempo**.

Para evitar isso, requisições condicionais podem ser usadas: adicionando o cabeçalho If-None-Match com o valor especial "*", que representa **qualquer** ETag.
A requisição só terá sucesso **se o recurso ainda não existir**:



If-None-Match só funcionará com servidores compatíveis com HTTP/1.1 (ou superior).
Se não tiver certeza se o servidor será compatível, primeiro emita uma requisição HEAD ao recurso para verificar.

Conclusão

Requisições condicionais são um recurso fundamental do HTTP e permitem a construção de aplicações eficientes e complexas.
Para cache ou retomada de downloads, o único trabalho necessário por parte dos administradores da web é **configurar corretamente o servidor**; definir ETags corretamente em alguns ambientes pode ser complicado.
Uma vez feito, o navegador enviará as requisições condicionais esperadas automaticamente.

Para **mecanismos de bloqueio**, o oposto é verdadeiro: os **desenvolvedores web** precisam emitir as requisições com os cabeçalhos apropriados, enquanto os **administradores** podem na maior parte das vezes confiar na aplicação para realizar as verificações.